

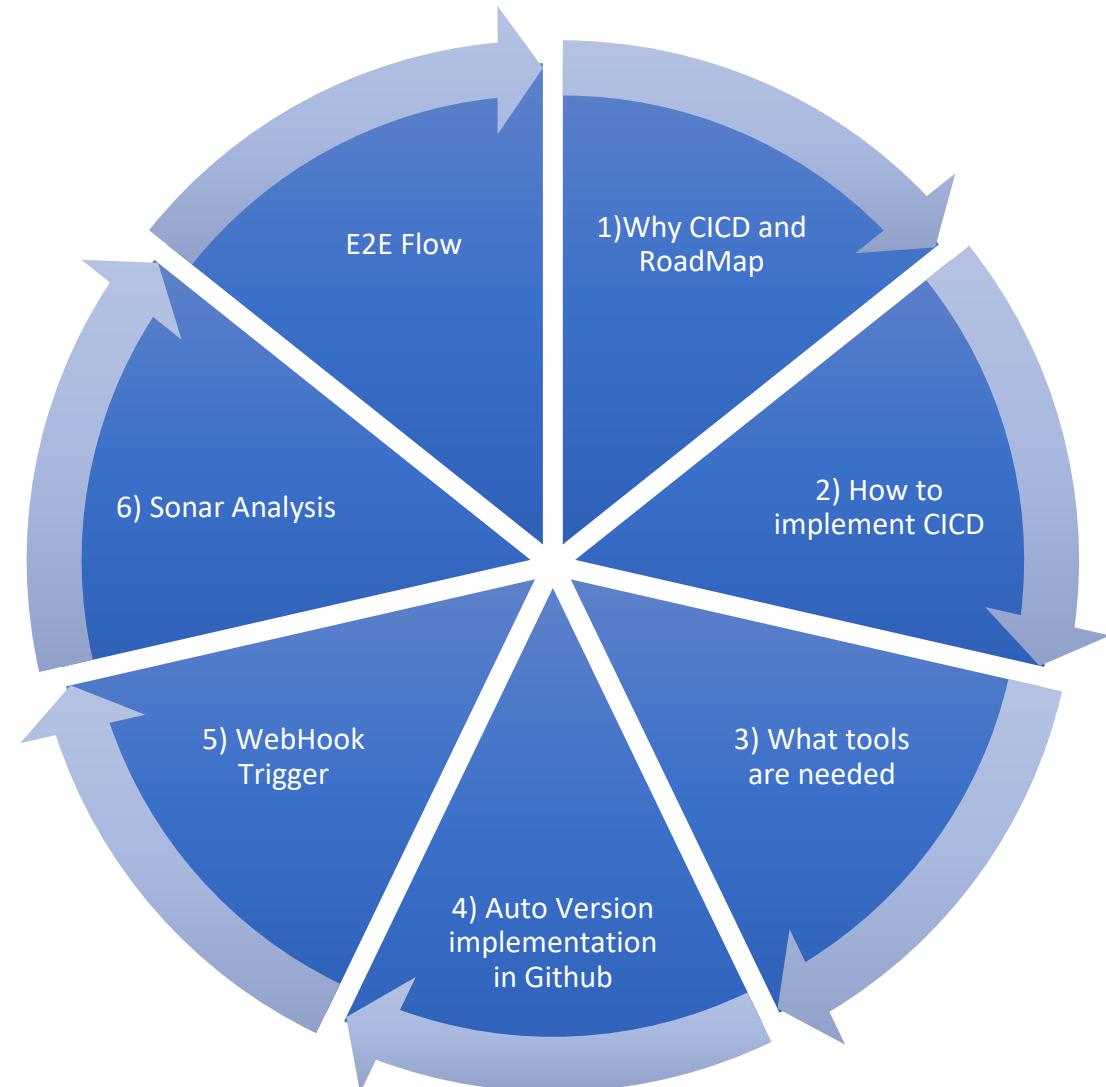
CICD End To End Flow

DevOps - CI/CD



By Praveen Singampalli

CICD discussion:





KubeSphere Container Platform

splunk®



Deployment
Service/Ingress



Developer



Monitor and Operate



Deploy to Kubernetes



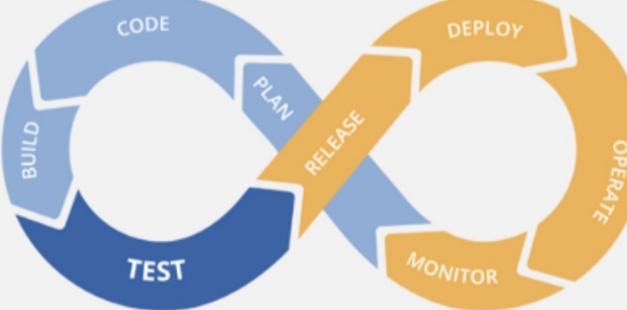
Build and Push Image



docker



JFrog



Source Code



Repository



Unit Test



Static Code Analysis



Build

JUnit

sonarqube

Jenkins

Maven

Different types of Jenkins Pipeline

- ***Declarative Pipeline Syntax***

The declarative syntax is a new feature that uses code for the pipeline. It provides a limited pre-defined structure. Thereby, it offers an easy & simple continuous delivery pipeline. Moreover, it uses a *pipeline block*.

- ***Scripted Pipeline Syntax***

Unlike declarative syntax, the *scripted pipeline syntax* is the old traditional way to write the *Jenkinsfile* on Jenkins web UI. Moreover, it strictly follows the groovy syntax and helps to develop a complex pipeline as code.

```
pipeline {  
    agent any  
    stages {  
        stage('One') {  
            steps {  
                echo 'Hi, welcome to pipeline demo...'  
            }  
        }  
        stage('Two') {  
            steps {  
                echo('Sample testing of Stage 2')  
            }  
        }  
        stage('Three') {  
            steps {  
                echo 'Thanks for using Jenkins Pipeline'  
            }  
        }  
    }  
}
```

Declarative Pipeline

```
pipeline {  
    agent {  
        // executes on an executor with the label 'some-label' or 'docker'  
        label "some-label || docker"  
    }  
  
    stages {  
        stage("foo") {  
            steps {  
                // variable assignment (other than environment variables) can only  
                // complex global variables (with properties or methods) can only  
                // env variables can also be set within a script block  
                script {  
                    foo = docker.image('ubuntu')  
                    env.bar = "${foo.imageName()}"  
                    echo "foo: ${foo.imageName()}"  
                }  
            }  
        }  
        stage("bar") {  
            steps{  
                echo "bar: ${env.bar}"  
                echo "foo: ${foo.imageName()}"  
            }  
        }  
    }  
}
```

Scripted Pipeline

```
node {  
  
    git url: 'https://github.com/jfrogdev/project-examples.git'  
  
    // Get Artifactory server instance, defined in the Artifactory Plugin  
    def server = Artifactory.server "SERVER_ID"  
  
    // Read the upload spec and upload files to Artifactory.  
    def downloadSpec =  
        '''{  
        "files": [  
            {  
                "pattern": "libs-snapshot-local/*.zip",  
                "target": "dependencies/",  
                "props": "p1=v1;p2=v2"  
            }  
        ]  
    }'''  
  
    def buildInfo1 = server.download spec: downloadSpec  
  
    // Read the upload spec which was downloaded from github.  
    def uploadSpec =  
        '''{  
        "files": [  
            {  
                "pattern": "resources/Kermit.*",  
                "target": "libs-snapshot-local",  
                "props": "p1=v1;p2=v2"  
            },  
            {  
                "pattern": "resources/Frogger.*",  
                "target": "libs-snapshot-local"  
            }  
        ]  
    }'''
```

Scripted Pipeline



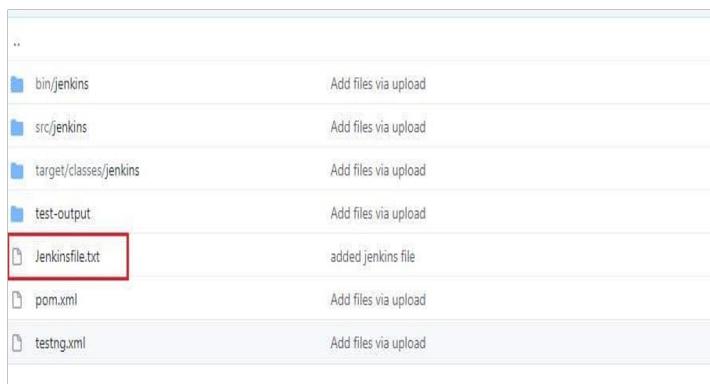
```
4 * stage('One') {  
5     steps {  
6         echo 'Hi, welcome to pipeline demo...'  
7     }  
8 }  
9 * stage('Two') {  
10    steps {  
11        echo('Sample testing of Stage 2')  
12    }  
13 }  
14 * stage('Three') {  
15    steps {  
16        echo 'Thanks for using Jenkins Pipeline'  
17    }  
18 }  
19 }  
20 }
```

Use Groovy Sandbox

Pipeline Syntax

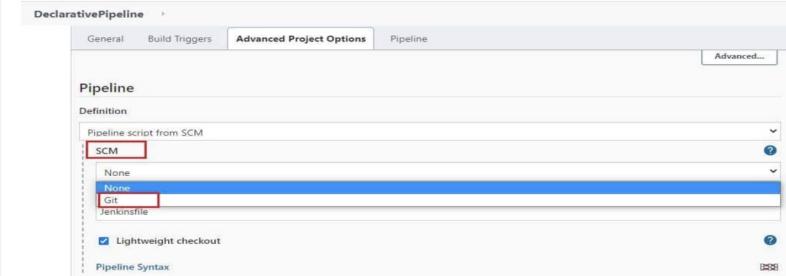
Save **Apply**

Declarative Pipeline(Jenkinsfile add to git repo)



..

- bin/jenkins Add files via upload
- src/jenkins Add files via upload
- target/classes/jenkins Add files via upload
- test-output Add files via upload
- Jenkinsfile.txt added jenkins file **Red Box**
- pom.xml Add files via upload
- testng.xml Add files via upload



DeclarativePipeline

General Build Triggers Advanced Project Options Pipeline

Advanced...

Pipeline

Definition

Pipeline script from SCM

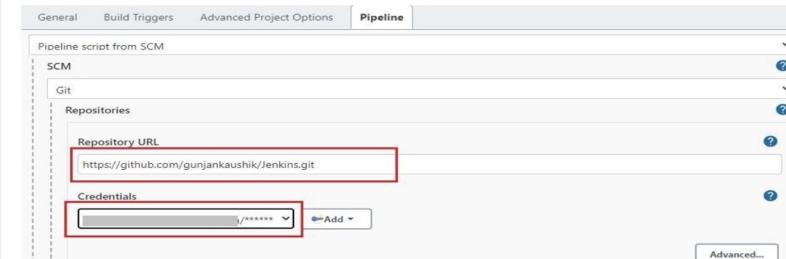
SCM

- None
- None
- Git **Red Box**
- Jenkinsfile

Lightweight checkout

Pipeline Syntax

3. Now, you will get an option to input your *Repository URL* and *credentials*.



General Build Triggers Advanced Project Options **Pipeline**

Pipeline script from SCM

SCM

Git

Repositories

Repository URL **Red Box**

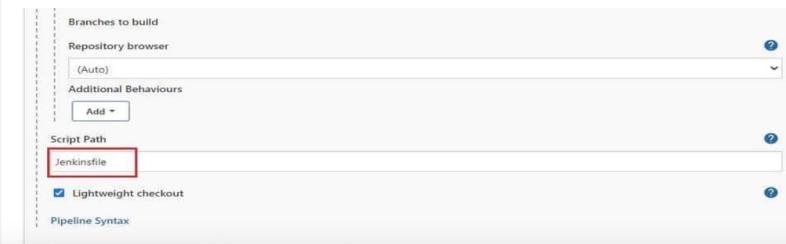
https://github.com/gunjanikaushik/Jenkins.git

Credentials **Red Box**

/***** Add

Advanced...

4. Next, you may set the *branch* or let it be blank for any branch. In the script path, you need to write the *Jenkinsfile* name that exists in your repository. Click on *Save*, and there you go, your declarative pipeline is ready for use.



Branches to build

Repository browser

(Auto)

Additional Behaviours

Add

Script Path **Red Box**

Jenkinsfile

Lightweight checkout

Pipeline Syntax

JenkinsFile Data

Key concepts of Jenkinsfile

- **Pipeline** - A pipeline is a set of instructions that includes the processes of continuous delivery. For example, creating an application, testing it, and deploying the same. Moreover, it is a critical element in declarative pipeline syntax, which is a collection of all stages in a Jenkinsfile. We declare different stages and steps in this block.
- **pipeline{ } Node** - A node is a key element in scripted pipeline syntax. Moreover, it acts as a machine in Jenkins that executes the Pipeline.
- **node{ } Stage** - A stage consists of a set of processes that the Pipeline executes. Additionally, the tasks are divided in each stage, implying that there can be multiple stages within a Pipeline. The below snippet shows the different stages that one can define in a Pipeline.
- **Steps** - A step in Jenkins defines what we have to do at a particular step in the process. There can be a series of steps within the stage. Moreover, whatever step we define in a stage would be executed within it.
- **Agent** - An agent is a directive that enables the users to execute multiple projects within the same Jenkins instance by distributing the load. Moreover, we assign an executor to the build through an agent. You can either use a single agent for the entire pipeline or use a distinct agent for the different stages of the pipeline. Subsequently, some of the parameters used with agents are -
 - **Any**- Any of the available agents execute the pipeline.
 - **None**- It is used at the pipeline root and implies no global agent, but each stage must specify its own agent.
 - **Label**- The labeled agent is used to execute the pipeline or the specific stage.
 - **Docker**- One can use the Docker images as the execution environment & specifying the agent as docker.

Merge Request from dev to master Branch

Child Pipeline -

DevTeam ↗ Code Commit ↗ To GitHub ↗ Triggers Child Pipeline (WebHook) ↗ Sonar/Blackduck/Fortify(Stage 1 - In Parallel) ↗ Creates the Merge Request(Stage 2 – Sends Email to Approver)

Master Pipeline -

Merge request gets approved ↗ Master Pipeline Trigger(WebHook) ↗ Sonar/Blackduck/Fortify(Stage 1 - In Parallel) ↗ Build Code (Maven) ↗ Artifact Push (Jfrog) ↗ Pull Artifact(Ansible) ↗ Deployment to QA(OnPrem/Cloud) ↗ Performance testing/Integration testing ↗ Deployment to Syage ↗ Send Notification via Email

Webhook Setup

- In your project or group, on the left sidebar, select Settings > Webhooks.
- In URL, enter the URL of the webhook endpoint.
- In Secret token, enter the secret token to validate payloads.
- In the Trigger section, select the events to trigger the webhook

The screenshot shows the GitLab 'Integrations' settings page for a project named 'Kevin / Gitlab Webhook Test'. A red box highlights the 'URL' field containing the webhook endpoint URL. Another red box highlights the 'Trigger' section, which lists various events like Push events, Tag push events, Comments, Issues events, Confidential Issues events, Merge Request events, Jobs events, Pipeline events, and Wiki Page events. At the bottom of the left panel, there is an 'Add Webhook' button. The right panel shows the 'Build Triggers' configuration, also with a red box highlighting the 'Trigger' section. It includes options for triggering builds remotely, after other projects, periodically, or when changes are pushed to GitHub. A specific trigger is selected for 'Build when a change is pushed to GitLab' with the URL 'http:// /project/Example%20Job'. The 'Enabled GitLab triggers' section has checkboxes for Push Events (checked), Merge Request Events (checked), and Rebuild open Merge Requests (set to 'Never'). The 'Comments' section has a checked checkbox. The 'Comment for triggering a build' field contains the text 'Jenkins please retry a build'. An 'Advanced...' button is visible at the bottom right of the triggers section.

Jenkins Master slave Config

1

Welcome to Jenkins!

Please [create new jobs](#) to get started.

ENABLE AUTO REFRESH

New Item
People
Build History
Manage Jenkins **←**
My Views
Credentials

Build Queue
No builds in the queue.

Build Executor Status
1 Idle
2 Idle

3

Back to Dashboard
Manage Jenkins
New Node **←**
Configure

Build Queue
No builds in the queue.

Build Executor Status
1 Idle
2 Idle

Node name:

Permanent Agent
Adds a plain, permanent agent to Jenkins. This is called "permanent" because Jenkins doesn't provide higher level of integration with these agents, such as dynamic provisioning. Select this type if no other agent types apply — for example such as when you are adding a physical computer, virtual machines managed outside Jenkins, etc.

OK

5

Back to List
Status
Delete Agent
Configure
Build History
Load Statistics
Log

Build Executor Status

Agent Slave1

Connect agent to Jenkins one of these ways:

- Launch** Launch agent from browser
- Run from agent command line:
`java -jar slave.jar -jnlpUrl http://localhost:8080/computer/Slave1/slave-agent.jnlp -secret 19385e54b9539fd6272df525dc9e7687efea162e44a83d918ca918294eb9f8e`

Projects tied to Slave1

None

2

Load Statistics
Check your resource utilization and see if you need more computers for your builds.

Jenkins CLI
Access/manage Jenkins from your shell, or from your script.

Script Console
Executes arbitrary script for administration/trouble-shooting/diagnostics.

Manage Nodes **←**
Add, remove, control and monitor the various nodes that Jenkins runs jobs on.

About Jenkins
See the version and license information.

4

Back to Dashboard
Manage Jenkins
New Node
Configure

Build Queue
No builds in the queue.

Build Executor Status
1 Idle
2 Idle

Name: Slave1
Description:
of executors: 1
Remote root directory: **Remote directory is mandatory**
Labels:
Usage: Use this node as much as possible
Launch method: Launch agent via execution of command on the master
Launch command: **No launch command specified**
Availability: Keep this agent online as much as possible

Node Properties
Environment variables
Tool Locations
Save

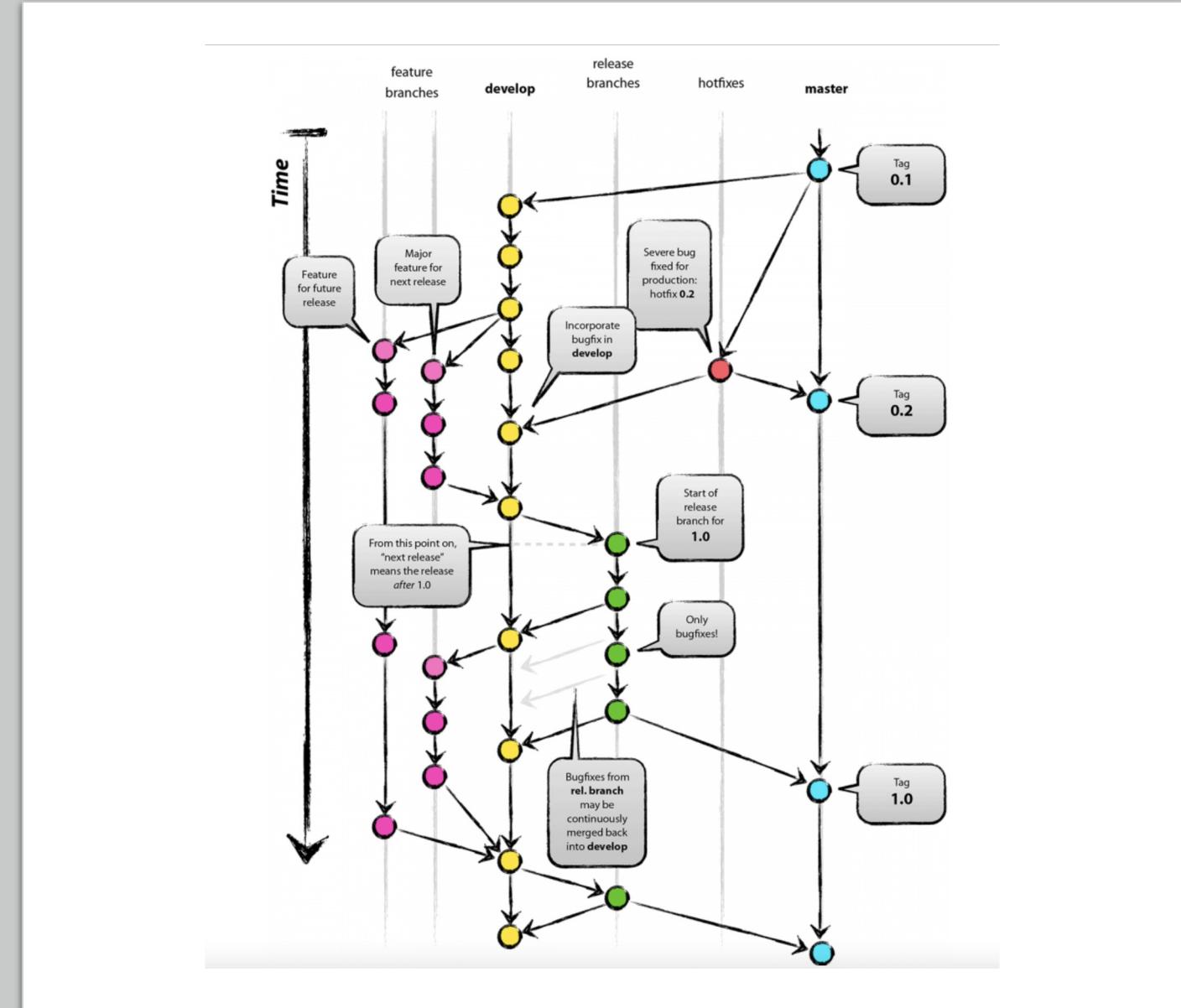
Trigger a pipeline from JenkinsFile

```
pipeline {
    agent {
        node {
            label 'master'
            customWorkspace "${env.JobPath}"
        }
    }
    stages {
        stage('Start') {
            steps {
                sh 'ls'
            }
        }
        stage ('Invoke_pipeline') {
            steps {
                build job: 'pipeline1', parameters: [
                    string(name: 'param1', value: "value1")
                ]
            }
        }
        stage('End') {
            steps {
                sh 'ls'
            }
        }
    }
}
```

```
pipeline {
    agent {
        docker {
            image 'node:6-alpine'
            args '-p 3000:3000 -p 5000:5000'
        }
    }
    environment {
        CI = 'true'
    }
    stages {
        stage('Build') {
            steps {
                sh 'npm install'
            }
        }
        stage('Test') {
            steps {
                sh './jenkins/scripts/test.sh'
            }
        }
        stage('Deliver for development') {
            when {
                branch 'development' ①
            }
            steps {
                sh './jenkins/scripts/deliver-for-development.sh'
                input message: 'Finished using the web site? (Click "Proceed" to continue)'
                sh './jenkins/scripts/kill.sh'
            }
        }
        stage('Deploy for production') {
            when {
                branch 'production' ②
            }
            steps {
                sh './jenkins/scripts/deploy-for-production.sh'
                input message: 'Finished using the web site? (Click "Proceed" to continue)'
                sh './jenkins/scripts/kill.sh'
            }
        }
    }
}
```

GitLab Branching Strategy

- **Master Branch** – This branch mainly deals with the prod deployment(Prod Pipeline)
- **Hotfix Branch** – This branch deals with any issues related to prod fixes (Bug/Feedback from customer)(Hotfix Pipeline)
- **Release Branch (Release Pipeline)** – The code from this branch is deployed in higher environments(QA/STAGE) except prod
- **Develop Branch (Dev Pipeline)**– Called as Dev branch and it deals with the changes done by developers for their own branch
- **Feature Branch** – Deals with the new features taken up for the sprint(We usually use to merge the feature to release branch and then deploy)



TOOLS INSTALLATION



SOFTWARES DOWNLOAD

1. Visual Studio Code [MAC AND WINDOWS]

<https://code.visualstudio.com/download>

2. Python Install. —> python3 --version

- <https://www.python.org/downloads/>

3. Install brew if it is not there in MAC [ONLY FOR MAC USERS]

- open the terminal and copy below command

STEP 1 - /bin/bash -c "\$(curl -fsSL <https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh>)"

STEP 2 - vim .zshrc

STEP 3 - now enter i to start edit mode and write

- export PATH=/opt/homebrew/bin:\$PATH
- STEP 4 - source ~/.zshrc

4. Terraform. <https://www.terraform.io/downloads> -> terraform -v

- ONLY IF TERRAFORM GIVES ERROR FOR MAC
- brew install kreuzwerker/taps/m1-terraform-provider-helper
- m1-terraform-provider-helper activate # (In case you have not activated the helper)
- m1-terraform-provider-helper install hashicorp/template -v v2.2.0 # Install and compile

5. AWS ACCESS

- <https://aws.amazon.com/console/>
- Sign In for FREE console
- It will ask for credit card details
- Keep the sign in details safe

6. Install aws cli - <https://docs.aws.amazon.com/cli/v1/userguide/install-macos.html>

- If PYTHON error comes on MAC then
- sudo /usr/local/bin/python3.11 awscli-bundle/install -i /usr/local/aws -b /usr/local/bin/aws

7. AWS CONFIGURE

- CREATE THE USER IN AWS -> SAVE THE ACCESS KEY AND SECRET KEY

9. GIT -> <https://git-scm.com/download/win>

10. Install Docker

- <https://docs.docker.com/desktop/install/mac-install/> -> MAC
- <https://docs.docker.com/desktop/install/windows-install/> -> WINDOWS

11. JAVA INSTALL :

- <https://jdk.java.net/18/>
- <https://www.digitalocean.com/community/tutorials/install-maven-mac-os>
- `sudo mv jdk-18.0.2.1.jdk /Library/Java/JavaVirtualMachines/`
- `JAVA_HOME="/Library/Java/JavaVirtualMachines/jdk-18.0.2.1.jdk/Contents/Home"`
- `PATH="${JAVA_HOME}/bin:${PATH}"`
- `export PATH`
- `java -version`

12. MAVEN INSTALL:

MAC - brew install maven

WINDOWS - <https://maven.apache.org/download.cgi>

13. MINIKUBE INSTALL:

- <https://minikube.sigs.k8s.io/docs/start/>

14. NODE/NPM INSTALL:

INSTALL NODE - <https://nodejs.org/en/download/>